

SOURCE CODE APPENDIX

```
----- CompTable.java -----
import java.util.*;

class CompTable {
    public CompTable(int arguments, int components, int characteristic) {
        dimin = arguments;
        dimout = components;
        if(characteristic < 3 || dimin < 1 || dimout < 1)
            System.out.println("CompTable:constructor 1: arguments out of
range.\n");
        p = characteristic;
        c = new int[Imath.ipow(p,dimin)];
    }

    public CompTable(int arguments, int components, int characteristic,
                      int[] data) {
        dimin = arguments;
        dimout = components;
        if(characteristic < 5 || dimin < 1 || dimout < 1 || dimout !=
data.length)
            System.out.println("CompTable:constructor 2: arguments out of
range.\n");
        p = characteristic;
        c = new int[Imath.ipow(p,dimin)];
        c = data;
    }

    public void Set(int index, int element) {
        // Assign a value to a function table element
        if(index >= 0 && index < c.length) c[index] = element;
    }

    public int Get(int index) {
        // Assign the value of a function table element
        if(index >= 0 && index < c.length)
            return c[index];
        else
            return -1;
    }

    public int ComputationStep(int argument) {
        // Compute a computation step using the function table
        if(argument < 0 || argument > c.length - 1) {
            System.out.println("CompTable:ComputationStep: argument out of
range.\n");
            return -1;
        }
        return c[argument];
    }

    public void UnivEncrypt(int[] encrypt, int[] decrypt,
                           int[][] permutation) {
        // Encrypt this mapping in its function table representation
        // using a univariate permutation in its function table representation
    }
}
```

00872742 121800

```
if(encrypt.length != dimout || decrypt.length != dimin) {
    System.out.println("CompTable:UnivEncrypt: Invalid arguments.\n");
    return;
}
// Note: every permutation must be of length p
// Every "encrypt" and "decrypt" integer must be either -1 or a
// reference to one of the permutations

// misc vars
int i, j, k, l, m;
int[] ind, tind;
int[] tc;
tc = new int[c.length];
int[] powers;

// vectorize and partially decrypt input placing result in tc array
ind = new int[dimin];
tind = new int[dimin];

for(i = 0; i < dimin; i++) ind[i] = 0;
for(i = 0; i < c.length; i++) {
    k = 0;
    for(j = dimin - 1; j >= 0; j--) {
        if(decrypt[j] >= 0 && decrypt[j] < dimin)
            tind[j] = permutation[decrypt[j]][ind[j]];
        else
            tind[j] = ind[j];
        k *= p;
        k += tind[j];
    }
    tc[k] = c[i];
    // increment index
    j = 0;
    do {
        ind[j] ++;
        if(ind[j] == p) ind[j] = 0;
        j++;
    } while(ind[j - 1] == 0 && j < dimin);
}

// vectorize and partially encrypt output of tc array
powers = new int[dimout];
k = 1;
for(i = 0; i < dimout; i++) {
    powers[i] = k;
    k *= p;
}
ind = new int[dimout];
for(i = 0; i < c.length; i++) {
    k = tc[i];
    m = 0;
    for(j = dimout - 1; j >= 0; j--) {
        l = k / powers[j];
        k -= l * powers[j];
        ind[j] = l;
        if(encrypt[j] >= dimin && encrypt[j] < dimin + dimout)
            ind[j] = permutation[encrypt[j]][ind[j]];
        m += powers[j] * ind[j];
    }
    tc[i] = m;
}
for(i = 0; i < c.length; i++) c[i] = tc[i];
```

09737742 "121800

```
}

public void MultivEncrypt(int[] vbl, int fbl[], int[] decrypt,
    int[] encrypt, int[][] permutation) {
// Encrypt this mapping in its function table representation
// using a multivariate permutation in its function table representation
// Note: the number of components in the variable blocks (sum vbl)
// must equal dimin. Similarly sum fbl must equal dimout.
// Each permutation is defined for one of the designated block
// lengths, and contains p^bl[i] elements.

// Misc. vars.
int i, j, k, l, m, sum, vdim, fdim;
int[] ind, tind;
int[] tc;
boolean ok;
tc = new int[c.length];

// Checks
if(vbl.length != decrypt.length || fbl.length != encrypt.length) {
    System.out.println(
        "CompTable:MultivEncrypt: Mismatching block lengths and\n" +
        "    encryption/decryption specifications.\n");
    return;
}
for(vdim = 0, i = 0; i < vbl.length; i++)    vdim += vbl[i];
for(fdim = 0, i = 0; i < fbl.length; i++)    fdim += fbl[i];
if(vdim != dimin || fdim != dimout) {
    System.out.println(
        "CompTable:MultivEncrypt: Block definitions do not match\n" +
        "    input or output dimensions for the target mapping.\n");
    return;
}

for(ok = true, i = 0; i < decrypt.length && ok; i++)
    ok = (decrypt[i] < permutation.length);
for(i = 0; i < encrypt.length && ok; i++)
    ok = (encrypt[i] < permutation.length);
if(!ok) {
    System.out.println(
        "CompTable:MultivEncrypt: Encryption/decryption key\n" +
        "    references out of range.\n");
    return;
}

// partially vectorize and partially decrypt input using
// an irregular base
int[] ivbase = new int[vbl.length];
int[] ivpow = new int[vbl.length];
for(k=1,i=0; i < vbl.length; i++) {
    ivbase[i] = k;
    ivpow[i] = Imath.ipow(p, vbl[i]);
    k *= ivpow[i];
}
int[] ifbase = new int[fbl.length];
int[] fvpow = new int[fbl.length];
for(k=1,i=0; i < fbl.length; i++) {
    ifbase[i] = k;
    fvpow[i] = Imath.ipow(p, fbl[i]);
    k *= fvpow[i];
}
```

09737742 121800

```
ind = new int[vbl.length]; // index variable blocks
tind = new int[vbl.length];

for(i = 0; i < vbl.length; i++) ind[i] = 0;
for(i = 0; i < c.length; i++) {
    k = 0; // start of current block
    for(j = vbl.length - 1; j >= 0; j--) { // Loop over variable blocks
        if(decrypt[j] >= 0 && decrypt[j] < permutation.length)
            // Convert part of index vector to one indexing integer
            tind[j] = permutation[decrypt[j]][ind[j]];
        else
            tind[j] = ind[j];
        k += ivpow[j];
    }
    tc[k] = c[i];

    // increment index
    j = 0;
    do {
        ind[j]++;
        if(ind[j] == ivpow[j]) ind[j] = 0;
        j++;
    } while(ind[j - 1] == 0 && j < vbl.length);
}

ind = new int[fbl.length];
for(i = 0; i < c.length; i++) {
    k = tc[i];
    m = 0;
    for(j = fbl.length - 1; j >= 0; j--) {
        l = k / ifbase[j];
        k -= l * ifbase[j];
        ind[j] = l;
        if(encrypt[j] >= 0 && encrypt[j] < fbl.length)
            ind[j] = permutation[encrypt[j]][ind[j]];
        m += ifbase[j] * ind[j];
    }
    tc[i] = m;
}
for(i = 0; i < c.length; i++) c[i] = tc[i];
}

public void ParamEncrypt(int[] vbl, int[] varin, int[] fbl, int[] varout,
    int[] decrypt, int[] encrypt, int[][] permutation) {
    // Encrypt using parametrized, multivariate key mappings
    // The parameters specify the following:
    //   vbl: Number of variables in each consecutive variable group
    //   varin: Array containing index of the
    //           variable group used as additional parameter per encryption.
    //           These variables come in addition to the original variables.
    //           The index is -1 if no parametrization occurs.
    //   fbl: Number of components in each consecutive component group
    //   varout: Array containing index of the
    //            variable group used as additional parameters per encryption.
    //            These variables come in addition to the original components.
    //            The index is -1 if no parametrization occurs.
    //   decrypt: Specifies whether a given block is to be left alone
    //             (value -1) or has an index indicating a decryption mapping
    //             in the permutations parameter.
    //   encrypt: Specifies whether a given block is to be left alone
    //            (value -1) or has an index indicating an encryption mapping
```

09737742 "121800

```
//          in the permutations parameter.
//  permutation: Specifies the all the permutations used for the
//          encryption.

// Misc. vars.
int i, j, k, l, m, n, vdim, fdim;
int[] ind, tind;
int[] tc;

boolean ok;
tc = new int[c.length];

// Checks
if(vbl.length != decrypt.length || fbl.length != encrypt.length ||
    vbl.length != varin.length || vbl.length != varout.length) {
    System.out.println(
        "CompTable:ParamEncrypt: Mismatching block lengths and\n" +
        "    encryption/decryption specifications.\n");
    return;
}
for(vdim = 0, i = 0; i < vbl.length; i++) vdim += vbl[i];
for(fdim = 0, i = 0; i < fbl.length; i++) fdim += fbl[i];
if(vdim != dimin || fdim != dimout) {
    System.out.println(
        "CompTable:ParamEncrypt: Block definitions do not match\n" +
        "    input or output dimensions for the target mapping.\n");
    return;
}

for(ok = true, i = 0; i < decrypt.length && ok; i++)
    ok = (decrypt[i] < permutation.length);
for(i = 0; i < encrypt.length && ok; i++)
    ok = (encrypt[i] < permutation.length);
if(!ok) {
    System.out.println(
        "CompTable:MultivEncrypt: Encryption/decryption key\n" +
        "    references out of range.\n");
    return;
}

// partially vectorize and partially decrypt input using
// an irregular base
int[] ivbase = new int[vbl.length];
int[] ivpow = new int[vbl.length];
for(k=1,i=0; i < vbl.length; i++) {
    ivbase[i] = k;
    ivpow[i] = Imath.ipow(p, vbl[i]);
    k *= ivpow[i];
}
int[] ifbase = new int[fbl.length];
int[] fvpow = new int[fbl.length];
for(k=1,i=0; i < fbl.length; i++) {
    ifbase[i] = k;
    fvpow[i] = Imath.ipow(p, fbl[i]);
    k *= fvpow[i];
}

ind = new int[vbl.length]; // index variable blocks
tind = new int[vbl.length];

for(i = 0; i < vbl.length; i++) ind[i] = 0;
for(i = 0; i < c.length; i++) {
```

09737742 "121800

```

k = 0; // start of current block
for(j = vbl.length - 1; j >= 0; j--) { // Loop over variable blocks
if(decrypt[j] >= 0 && decrypt[j] < permutation.length) {
    // As a convention, the additional parameters make up the
    // least significant bits of the lumped-together integer.
    if(varin[j] >= 0) {
        m = ind[j] * ivpow[varin[j]] + ind[varin[j]];
        tind[j] = permutation[decrypt[j]][m];
    } else
        tind[j] = permutation[decrypt[j]][ind[j]];
    } else
        tind[j] = ind[j];
    k *= ivpow[j];
    k += tind[j];
}
tc[k] = c[i];

// increment index
j = 0;
do {
    ind[j]++;
    if(ind[j] == ivpow[j]) ind[j] = 0;
    j++;
} while(ind[j - 1] == 0 && j < vbl.length);
}

for(i = 0; i < vbl.length; i++) ind[i] = 0;
int[] find = new int[fbl.length];
for(i = 0; i < c.length; i++) {
    k = tc[i];
    m = 0;
    for(j = fbl.length - 1; j >= 0; j--) {
        l = k / ifbase[j];
        k -= l * ifbase[j];
        find[j] = l;
        if(encrypt[j] >= 0 && encrypt[j] < fbl.length) {
            if(varout[j] >= 0) {
                n = find[j] * ivpow[varout[j]] + ind[varout[j]];
                find[j] = permutation[encrypt[j]][n];
            } else
                find[j] = permutation[encrypt[j]][find[j]];
            }
        m += ifbase[j] * find[j];
    }
    tc[i] = m;

    // increment index
    j = 0;
    do {
        ind[j]++;
        if(ind[j] == ivpow[j]) ind[j] = 0;
        j++;
    } while(ind[j - 1] == 0 && j < vbl.length);
}

for(i = 0; i < c.length; i++) c[i] = tc[i];
}

private int[] c;
private int p, dimin, dimout;
}
```

0973742 121800

09737742 121800

```
---- Imath.java ---- (minor code used by CompTable.java)
public final class Imath extends Object {

    public static int ipow(int n, int e) {
        // Exponentiation using the square-and-multiply algorithm
        int prod = 1, k = n;
        while (e > 0) {
            if( (e&1) == 1) prod *= k;
            e >>= 1;
            k *= k;
        }
        return (n == 0 ? 0: prod);
    }
}
```

008727424E/60


```
import java.io.*;
import java.text.*;
import java.util.*;
import Fpoly.*;
```

```

class MappingTempl {
    public MappingTempl(int vars, int comps) {
        // Declare a mapping template, which contains enough information about
        // the mapping to enable correct encryption/"decryption" of any of its
        // variables and/or function components.
        d = vars;
        e = comps;
        name = new int[e][];
    }

    public void setmappingtempl(int[][] vname) {
        // Fix the naming pattern and template for each mapping component.
        // vname is a ragged array, containing the naming pattern, which has
        // the following format:
        //     - The first index refers to the mapping component.
        //     - The second index refers to the variable of that component.
        //     - The array entry itself contains the variable's index--that is
name
        //         as it is referenced in the complete mapping
        // Check to see if vname has right number of variables and components.
        if(vname.length != e) {
            System.out.println("setmappingtempl: Mismatched no. of function"
                +" components.");
            return;
        }
        int j = 0;
// A check is missing here, but this should not be a problem for
// correctly specified arguments
        name = vname;
    }

    public int noofvars() {
        // Return total number of variables
        return d;
    }

    public int noofcomponents() {
        // Return total number of components
        return e;
    }

// d is the number of variables, e the number of function components
    public int d, e;
    // name keeps track of the variable names, enabling correct encryption.
    // Note that name is in general a ragged array.
    public int[][] name;
}

```

```
----- Univkeys.java -----
import java.io.*;
import java.text.*;
import java.util.*;

class Univkeys {
    public Univkeys(int modulus) {
        // Declare asymmetric secret key pair
        p = modulus;
        e = new Fpoly(1, p);
        d = new Fpoly(1, p);
    }

    public void generate(int seed) {
        // Generate asymmetric secret key pair
        int i, j, k, n, tmp;
        int[] efunc = new int[p];
        int[] dfunc = new int[p];
        int[][] a = new int[p][p];
        int[] factorial = new int[p];
        // Initialize pseudo random number generator
        Random prng = new Random(seed);
        // Initialize permutation and lagrange function data
        for(i = 0; i < p; i++) {
            //Mark the inverse function's table entries to ensure bijectivity
            dfunc[i] = -1;
            efunc[i] = 0;
            for(j = 1; j < p; j++) a[i][j] = 0; // Lagrange function data
            a[i][0] = 1; // Lagrange function data
            factorial[i] = 1;
        }
        // Generate the permutations
        for(i = 0; i < p; i++) {
            do {
                n = prng.nextInt();
                if(n < 0) n = - n;
                n %= p;
            } while(dfunc[n] != -1);
            efunc[i] = n;
            dfunc[n] = i;
        }

        // Interpolate symbolically to find the resultant mappings.
        // Lagrange interpolation is used.
        // First is precomputation of the a_i(x) polynomials
        for(k = 0; k < p; k++) {
            for(i = 0; i < p; i++) {
                if(i != k) {
                    for(j = p-1; j > 0; j--) {
                        tmp = e.table.fsub[0][k];
                        tmp = e.table.fmul[tmp][a[i][j]];
                        a[i][j] = e.table.fadd[tmp][a[i][j-1]];
                    }
                    a[i][0] = e.table.fmul[e.table.fsub[0][k]][a[i][0]];
                    factorial[i] = e.table.fmul[factorial[i]][e.table.fsub[i][k]];
                }
            }
        }
        // Invert denominators and apply expressions
        for(i = 0; i < p; i++) {
            factorial[i] = e.table.finv[factorial[i]];
        }
    }
}
```

09737742-121800

```
        for(j = 0; j < p; j++) a[i][j] =
e.table.fmul[a[i][j]][factorial[i]];
    }

    // Compute the keys
    for(i = 0; i < p; i++)
        for(j = 0; j < p; j++) {
            n = e.table.fmul[a[i][j]][efunc[i]];
            e.setcoef(j, e.table.fadd[n][e.coef(j)]);
            n = e.table.fmul[a[i][j]][dfunc[i]];
            d.setcoef(j, e.table.fadd[n][d.coef(j)]);
        }
    }

    public void printkeys() {
        System.out.println("Encryption key is: e= "+e.print());
        System.out.println("Decryption key is: d= "+d.print());
    }

    public void identity() {
        // A convenience function to ensure well-defined encryption templates
        for(int i = 2; i < e.noofcoeffs(); i++) {
            e.setcoef(i,0);
            d.setcoef(i,0);
        }
        e.setcoef(0,0);
        d.setcoef(0,0);
        e.setcoef(1,1);
        d.setcoef(1,1);
    }

    // p is the order of the finite field over which the keys are defined
    public int p;
    // e is the encryption key, and d the decryption key
    public Fpoly e, d;
    // All data are public, as this must in any case used as a private
    component
    // of another object
}
```

09/27/10 12:13:00

```
----- UnivEncTempl.java -----
import java.io.*;
import java.text.*;
import java.util.*;
import Univkeys.*;

class UnivEncTempl {
    public UnivEncTempl(int vars, int comps, int modulus) {
        // Declare a univariate encryption template for a mapping with d
        // variables and e function components over the integers modulo p.
        // p must be a prime number.
        d = vars;
        e = comps;
        p = modulus;
        crypt = new int[d + e];
        equiv = new int[d + e];
        keyused = new int[d + e];
        ready = false;
    }

    public void setencpattern(int[] encrflag, int[] keyequiv) {
        // Define encryption pattern for this template
        if(encrflag.length != d + e || keyequiv.length != d + e) {
            System.out.println("setencpattern: Encryption/key equivalence"
                + " arrays have wrong length.");
            return;
        }
        // The format for the encryption flag array is as follows:
        // +1: Encrypt the component or variable in question
        // 0: Do nothing
        // -1: Decrypt the component or variable in question
        // The format for the keyequivalenc flag array is as follows:
        // For the i'th entry:
        // -1: No equivalence with other key pair required. The first
        //     element of the array must always have this value.
        // 0 <= j < i: Equivalence with other key pair required.
        // So before proceeding, check the contents of the arrays to make sure
        // they are valid.
        for(int i = 0; i < d + e; i++)
            if(encrflag[i] < -1 || encrflag[i] > 1
                || keyequiv[i] < -1 || keyequiv[i] >= i) {
                System.out.println("setencpattern: Encryption/key equivalence"
                    + " arrays contain bad values.");
                return;
            }
        // Prepare first set of keys so that the pattern can be used
        int i, j;
        int k = 0;
        // Count number of individual key pairs needed and also generate
        // inverse reference from variables/components to keys they "use".
        // The format of this inverse array is as follows:
        // -1: No reference
        // 0 .. pairs-1: the key pair to use
        for(i = 0; i < d + e; i++)
            if(encrflag[i] != 0)
                if(keyequiv[i] == -1) {
                    keyused[i] = k;
                    k++;
                } else
                    keyused[i] = keyused[keyequiv[i]];
        else
            keyused[i] = -1;
    }
}
```

05737742 121600


```

        // If variable no. m.name[i][j] is used encrypted, then
        // it must be decrypted prior to application of a mapping.
        keytmp[k] = new Fpoly(key[keyused[m.name[i][j]]].d);
        k++;
    } else if(crypt[m.name[i][j]] == -1) {
        // If variable no. m.name[i][j] is used decrypted (NOT in
        // plaintext), then it must be encrypted prior to
        // application of a mapping.
        keytmp[k] = new Fpoly(key[keyused[m.name[i][j]]].e);
        k++;
    }
    result[i].composewith(keytmp, tv, m.name);
}

// Second compose the partially encrypted function with its
// encryption key if that has been chosen, otherwise decrypt.
if(crypt[d + i] == 1) {
    // If component no. i is used in encrypted form, encrypt.
    keytmp = new Fpoly[1];
    keytmp[0] = new Fpoly(result[i]);
    tv2 = new int[1];
    tv2[0] = -1;
    v2 = new int[1][result[i].dimension()];
    for(j = 0; j < v2[0].length; j++) {
        v2[0][j] = j;
    }
    keytmp2 = new Fpoly(key[keyused[d + i]].e);
    keytmp2.composewith(keytmp, tv2, v2);
    result[i] = keytmp2;
} else if(crypt[d + i] == -1) {
    keytmp = new Fpoly[1];
    keytmp[0] = new Fpoly(result[i]);
    tv2 = new int[1];
    tv2[0] = -1;
    v2 = new int[1][result[i].dimension()];
    for(j = 0; j < v2[0].length; j++) {
        v2[0][j] = j;
    }
    keytmp2 = new Fpoly(key[keyused[d + i]].d);
    keytmp2.composewith(keytmp, tv2, v2);
    result[i] = keytmp2;
}
}
}

public int noofvars() {
    // Return number of variables.
    return d;
}

public int noofcomponents() {
    // Return number of function components.
    return e;
}

public int noofkeys() {
    // Return number of distinct key pairs.
    return pairs;
}

// Note: there is no decryption method: i.e. no method of undoing the
// partial encryption process, as unambiguous decryption is

```

// mostly not possible.

```
public void printkeys() {
// A method for printing keys
// Used mainly for debugging
    for(int i = 0; i < key.length; i++) {
        System.out.println("Encryption key no. "+i+" = "
            +key[i].e.prettyprint());
        System.out.println("Decryption key no. "+i+" = "
            +key[i].d.prettyprint());
    }
}

private boolean ready;
// Number of variables, number of mapping components, modulus
private int d, e, p;
// Encryption/decryption and equivalence relations
private int[] crypt, equiv;
// An inverse reference array from a variable/component to a key pair
private int[] keyused;
// Number of key pairs
private int pairs;
// The key pairs
//private Univkeys[] key;
public Univkeys[] key;
}

---- Imath.java ----
public final class Imath extends Object {
// Just a simple class to implement and export useful mischellany.

    public static int ipow(int n, int e) {
// Exponentiation using the square-and-multiply algorithm
        int prod = 1, k = n;
        while (e > 0) {
            if( (e&1) == 1) prod *= k;
            e >>= 1;
            k *= k;
        }
        return (n == 0 ? 0: prod);
    }
}
```

09737742 121500

```

class Ftable {
    public Ftable(int p) {    int i,j,k;
        fmul = new int[p][p];
        fdiv = new int[p][p];
        fpow = new int[p][p];
        fadd = new int[p][p];
        fsub = new int[p][p];
        finv = new int[p];
        for(i = 0; i < p; i++) {
            k = 1;
            for(j = 0; j < p; j++) {
                fmul[i][j] = (i*j) % p;
                fpow[i][j] = k;
                k *= i;
                k %= p;
                fadd[i][j] = (i+j) % p;
                fsub[i][j] = (p+i-j) % p;
                if((j*i) % p == 1)    finv[i]=j;
            }
            fdiv[i][0] = -1;
        }
        for(i = 0; i < p; i++)
            for(j = 1; j < p; j++)
                fdiv[i][j] = fmul[i][finv[j]];
        fc = p;
    }

    public void checktable() {
        int i,j;
        System.out.println("Multiplication table");
        for(i = 0; i < fc; i++) {
            for(j = 0; j < fc; j++) {
                System.out.print(fmul[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println("Division table");
        for(i = 0; i < fc; i++) {
            for(j = 0; j < fc; j++) {
                System.out.print(fdiv[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println("Addition table");
        for(i = 0; i < fc; i++) {
            for(j = 0; j < fc; j++) {
                System.out.print(fadd[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println("Subtraction table");
        for(i = 0; i < fc; i++) {
            for(j = 0; j < fc; j++) {
                System.out.print(fsub[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println("Exponentiation table");
        for(i = 0; i < fc; i++) {
            for(j = 0; j < fc; j++) {
                System.out.print(fpow[i][j]+" ");
            }
        }
    }
}

```



```
    }
    System.out.println();
}
System.out.println("Inversion table");
for(i = 0; i < fc; i++) {
    System.out.print(finv[i]+" ");
}
}

public    int[][] fmul, fdiv, fpow, fadd, fsub;
public    int[] finv;
int fc;
}
```

09737742 121800

```

---- Fpoly.java ----
import java.util.*;

class Fpoly {
    public Fpoly(int dim, int characteristic) {
        // Initialize a polynomial with d=dim variables over a finite field
        // with p=characteristic elements. p must be a prime number.
        d = dim;
        p = characteristic;
        c = new int[Imath.ipow(p,d)];
        // Also generate a custom table with precomputed results to
        // (hopefully) speed up computations
        table = new Ftable(p);
    }

    public Fpoly(int dim, int characteristic, int[] data) {
        // Initialize a polynomial as above, but now also supplying the
        // coefficient data.
        d = dim;
        p = characteristic;
        c = new int[Imath.ipow(p,d)];
        for(int i = 0; i < Imath.ipow(p,d); i++) {
            if(data[i] < 0 || data[i] >= p) {
                System.out.println("Warning! Mismatched fields for polynomial
data!");
                c[i] = Math.abs(data[i]) % p;
            }
            else
                c[i] = data[i];
        }
        table = new Ftable(p);
    }

    public Fpoly(Fpoly b) {
        // Initialize a new polynomial equal to b.
        d = b.dimension();
        p = b.over();
        int l = Imath.ipow(p,d);
        c = new int[l];
        for(int i = 0; i < l; i++)
            c[i] = b.coeff(i);
        table = new Ftable(p);
    }

    public int dimension() {return d;}
    // Return the number of variables in this polynomial.

    public int over() {return p;}
    // Return the order of the finite field over which this polynomial
    // has been constructed.

    public int coeff(int index) {return c[index];}
    // Return the index'th coefficient of this polynomial.
    // Note: the index is always one-dimensional regardless of the
    // number of variables involved

    public int noofcoeffs() {return c.length;}
    // Return the number of coefficients

    private void consistency() {
        // A simple consistency check. Only used for debugging.
        int l = noofcoeffs();
    }
}

```

09737742 121800

```
if(l != c.length)
    System.out.println("Consistency: coefficient array has bad length.");
for(int i = 0; i < l; i++)
    if(c[i] < 0 || c[i] >= p)
        System.out.println("Consistency: coefficient no. "+i+" out of
range.");
}

public void setcoef(int i,int nc) {
// Set the value of a coefficient.
    if(nc < 0 || nc >= p) {
        System.out.println("setcoef: Coefficient out of range.");
        return;
    }
    c[i] = nc;
}

public void setequalto(Fpoly b) {
// Set this polynomial equal to polynomial b, assuming that
// this polynomial has the same number of variables as b and
// is defined over the same field.
    int l = Imath.ipow(p,d);
    if(p == b.over() && d == b.dimension()) {
        for(int i = 0; i < l; i++)
            c[i] = b.coeff(i);
    } else {
        System.out.println("setequalto: Mismatched polynomials.");
    }
}

public void changeto(Fpoly b) {
// Set this polynomial equal to b in all respects.
    d = b.dimension();
    p = b.over();
    int l = b.noofcoeffs();
    c = new int[l];
    for(int i = 0; i < l; i++)
        c[i] = b.coeff(i);
    table = new Ftable(p);
}

public void add(Fpoly q) {
// Add polynomial q to this polynomial, but only if q is defined
// over the same field as this polynomial.
// Prerequisite: q must have dimension less than the polynomial
// it is being added to.
    int qp = q.over();
    int qd = q.dimension();
    int ql = q.noofcoeffs();
    if(p == qp && d >= qd) {
        for(int i = 0; i < ql; i++)
            c[i] = table.fadd[q.coeff(i)][c[i]];
    } else {
        System.out.println("Warning! Polynomials are not properly matched
for"
        +" addition.");
    }
}

int addexp(int a, int b) {
    if (a < 0 || b < 0) {
        System.out.println("Bad arguments in addexp. Operation ignored.\n");
    }
}
```

09737742 " 121800

```

    return -1;
}
int r = a + b;
while(r > p - 1) { r -= (p - 1);}
return r;
}

public void multiplyby(int[] mind, int mcoef, int[] overlap) {
    // Multiply by a monomial
    // It is assumed that the monomial has:
    //     - a coefficient in the integers mod p
    //     - exponent <= p-1 for all variables
    // The resulting polynomial has as its first d variables the
    // d variables of the original polynomial, and thereafter any
    // non-overlapping variables.
    // No consistency check is performed on overlap.
    int i, j, k, l; // general index variables
    int nd, nl; // new dimensions
    // Count total monomial variables - real overlaps
    l = 0;
    for(i = 0; i < overlap.length; i++)
        if(overlap[i] < 0 || overlap[i] > d-1) l++;
    nd = d + l;
    nl = Imath.ipow(p, nd);
    Fpoly r = new Fpoly(nd, p);
    int[] rind = new int[nd];
    int[] ind = new int[d];
    if (mind.length != overlap.length) {
        System.out.println("multiplyby: No. of vars in arguments not
consistent.");
        return;
    }
    if (mcoef < 0 || mcoef >= p) {
        System.out.println("multiplyby: Monomial coefficient out of range.");
        return;
    }
    // Initialize index vectors (to keep track of variables' exponents)
    for(i = 0; i < d; i++) ind[i] = 0;
    // Initialize some other stuff
    for(i = 0; i < nl; i++) r.setcoef(i, 0);
    // Let's start the computation
    for(i = 0; i < c.length; i++) {
        // Do a multiplication --- but only if c[i] != 0; it's no use
        otherwise
        if(c[i] != 0) {
            for(j = 0; j < d; j++) rind[j] = ind[j];
            k = 0;
            for(j = 0; j < mind.length; j++) {
                if (overlap[j] > -1 && overlap[j] < d)
                    rind[overlap[j]] = addexp(rind[overlap[j]], mind[j]);
                else {
                    rind[d+k] = mind[j]; // addition cannot be done here
                    k++;
                }
            }
            // Compute linear version of index for resulting polynomial
            k = rind[nd - 1];
            for(j = nd - 2; j >= 0 ; --j) {
                k *= p;
                k += rind[j];
            }
            // Change coefficient in result

```

008721" 247E760

```

        l = table.fadd[r.coef(k)][table.fmul[mcoef][c[i]]];
        r.setcoef(k, l);
    }
    // Increment index
    k = 0;
    do {
        ind[k]++;
        if(ind[k] > p - 1) ind[k] = 0;
        k++;
    } while(ind[k - 1] == 0 && k < d);
    }
    this.changeto(r);
}

public void times(Fpoly b, int[] v) {
    // First perform consistency check on v, the variable
    // correspondence list
    int i,j,k;
    if (v.length != b.dimension()) {
        System.out.println("times: Dimension of polynomial and overlap"
            +" vector don't match.");
        return;
    }
    // Also count number of "new" variables relative to d
    k = 0;
    for(i = 0; i < v.length; i++) {
        for(j = i + 1; j < v.length; j++)
            if (v[i] == v[j]) {
                System.out.println("times: Duplicate overlaps defined.");
                return;
            }
        if (v[i] > -1 && v[i] < d) k++;
    }
    // Second initialize the temporary variable used to store the result
    int nl, nd, bl;
    nd = d + b.dimension() - k; // Note: Inconsistencies may still occur!
    nl = Imath.ipow(p, nd);
    bl = b.noofcoeffs();
    Fpoly r = new Fpoly(nd,p);
    Fpoly tmp = new Fpoly(this);
    int[] ind = new int[b.dimension()];
    for(i = 0; i < b.dimension(); i++) {ind[i] = 0;}
    // The multiplication is broken down into multiplication
    // by individual monomials.
    for(i = 0; i < bl; i++) {
        if(b.coef(i) != 0) {
            tmp.multiplyby(ind,b.coef(i),v);
            r.add(tmp);
            tmp.changeto(this);
        }
    }
    // Increment index vector
    j = 0;
    do {
        ind[j]++;
        if(ind[j] > p - 1) {ind[j] = 0;}
        j++;
    } while (j < b.dimension() && ind[j-1] == 0);
    }
    this.changeto(r);
}

```

007742 121800


```
for(j = 0; j < d; j++) {
    if(tv[j] < 0)
        ind[-(1 + tv[j])] = lind[j]; // An entry is a substitution or...
    else
        rind[rv[j]] = lind[j];      // a variable.
}
tp = ind[nopol - 1];
for(j = nopol - 2; j >= 0; j--) {tp *= p; tp += ind[j];}
// Now we can actually do the substitution itself
for(j = 0; j < nd; j++) tind[j] = 0;
for(j = 0; j < nl; j++) {
    for(k = 0; k < nd; k++) tmpind[k] = addexp(rind[k], tind[k]);
    rp = tmpind[nd - 1];
    for(k = nd - 2; k >= 0; k--) {rp *= p; rp += tmpind[k];}
    r.setcoef(rp,
table.fadd[r.coef(rp)][table.fmul[t[tp].coef(j)][c[i]]]);
    k = 0;
    do {
        tind[k]++;
        if(tind[k] > p - 1) tind[k] = 0;
        k++;
    } while(k < nd && tind[k - 1] == 0);
}
// Update index vector for "this" polynomial
j = 0;
do {
    lind[j]++;
    if(lind[j] > p - 1) lind[j] = 0;
    j++;
} while(j < d && lind[j - 1] == 0);
}
this.changeto(r);
}

public void multiplyby(int b) {
// Multiply the entire polynomial by a constant
for(int i = 0; i < c.length; i++) c[i] = table.fmul[c[i]][b];
}

public int evaluate(int[] x) {
// Evaluate the value of this polynomial at x using a Horner-like
// algorithm. -1 is returned if an error occurs.
// First some checks on the input
if(x.length != d) {
    System.out.println("evaluate: Input vector has wrong dimension.");
    return -1;
}
// Begin computation by computing the individual monomials
int lead = 0;
int lfact = 1;
int i, k = 0;
int[] ind = new int[d];
int[] t = new int[c.length];
t[0] = 1;
ind[0] = 1; for(i = 1; i < d; i++) ind[i] = 0;
for(i = 1; i < c.length; i++) {
    t[i] = table.fmul[x[lead]][t[i - 1]fact]];
    k = 0;
    do {
        ind[k]++;
        if(ind[k] > p - 1) ind[k] = 0;
```

09737742 "121800

```

        k++;
    } while(k < d && ind[k - 1] == 0);
    if(k - 1 > lead) {
        lead++;
        lfact *= p;
    }
}
// Finish computation by multiplying each monomial with its
// corresponding coefficient and adding.
int sum = 0;
for(i = 0; i < c.length; i++)
    sum = table.fadd[sum][table.fmul[c[i]][t[i]]];
return sum;
}

public String print() {
// Printout routine that generates a string with a list of monomials
// starting with the monomial with highest total degree, and descending
// to the constant monomial. Monomials are written on the form
// cx1^e1x2^e2...xd^ed, where c is the constant, x1...xd variables, and
// e1...ed exponents.
String tmp = "";
int[] ex = new int[d];
int i, j;
for(i = 0; i < d; i++)
    ex[i] = p - 1;
for(i = c.length - 1; i >= 0; i--) {
    if(i < c.length - 1) tmp += "+";
    tmp += c[i];
    for(j = 0; j < d; j++)
        tmp += "x" + (j+1) + "^" + ex[j];
    j = 0;
    do {
        ex[j] = table.fsub[ex[j]][1];
        if(j > 0) tmp += "\n";
        j++;
    } while (j < d && ex[j-1] == p-1);
}
return tmp;
}

public String prettyprint() {
// Printout routine similar to Fpoly.print() except that every time
// the last variable has its exponent "reset" to p-1 during the printing,
// a newline character is inserted.
String tmp = "";
int[] ex = new int[d];
int i, j;
for(i = 0; i < d; i++)
    ex[i] = p - 1;
for(i = c.length-1; i >= 0; i--) {
    if(i < c.length-1) tmp += "+";
    tmp += c[i];
    for(j = 0; j < d; j++)
        tmp += "x" + (j+1) + "^" + ex[j];
    j = 0;
    do {
        ex[j] = table.fsub[ex[j]][1];
        j++;
    } while (j < d && ex[j-1] == p-1);
    if(j > 1) tmp += "\n";
}
}

```

09737742 121800


```
        return tmp;
    }

    private    int[] c;
    private    int p,d;
        Ftable table;
    }
```

09737742 121800

----- Multikeys.java -----

```
import java.io.*;
import java.text.*;
import java.util.*;

class Multikeys {
//IMPORTANT: This object's implementation is INCOMPLETE. It is not
intended to
//provide the multivariate key functionality in its present form.
// NOTE: The completion of this class is analogous to Univkeys,
// with differences having to do with number of
// variables block sizes etc.
    public Multikeys(int modulus, int blocksize) {
        p = modulus;
        c = blocksize;
        e = new Fpoly(c, p);
        d = new Fpoly(c, p);
    }

    public void generate(int seed) {
        // Generate asymmetric secret key pair
        int i, j, k, n, tmp;
        int cl = e.noofcoeffs();
        int[][] efunc = new int[cl][c+1];
        int[][] dfunc = new int[cl][c+1];
        int[][] a = new int[p][p];
        int[] suma = new int[p];
        int[] factorial = new int[p];
        // Precompute a_n(y) functions
        for(i = 0; i < p; i++) {
            for(j = 1; j < p; j++) a[i][j] = 0; // Lagrange function data
            a[i][0] = 1; // Lagrange function data
            factorial[i] = 1;
        }
        for(k = 0; k < p; k++) {
            for(i = 0; i < p; i++) {
                if(i != k) {
                    for(j = 1; j < p; j++) {
                        tmp = e.table.fmul[e.table.fsub[0][k]][a[i][j]];
                        suma[j] = e.table.fadd[tmp][a[i][j-1]];
                    }
                    suma[0] = e.table.fmul[e.table.fsub[0][k]][a[i][0]];
                    factorial[i] = e.table.fmul[factorial[i]][e.table.fsub[i][k]];
                }
            }
        }
        // Initialize pseudo random number generator
        Random prng = new Random(seed);
        // Initialize permutation and lagrange function data
        for(i = 0; i < cl; i++) {
            //Mark the inverse function's table entries to ensure bijectivity
            dfunc[i][0] = -1;
            efunc[i][0] = 0;
        }
        // Generate the permutations
        for(i = 0; i < p; i++) {
            do {
                n = prng.nextInt();
                if(n < 0) n = - n;
                n %= p;
            } while(dfunc[n][0] != -1);
            efunc[i][0] = n;
        }
    }
}
```

0037742 121800

```
        dfunc[n][0] = i;
    }
}

int p, c;
Fpoly e, d;
}
```

09737742 121800